Kubernetes

Mithun Technologies

Kubernetes

Agenda

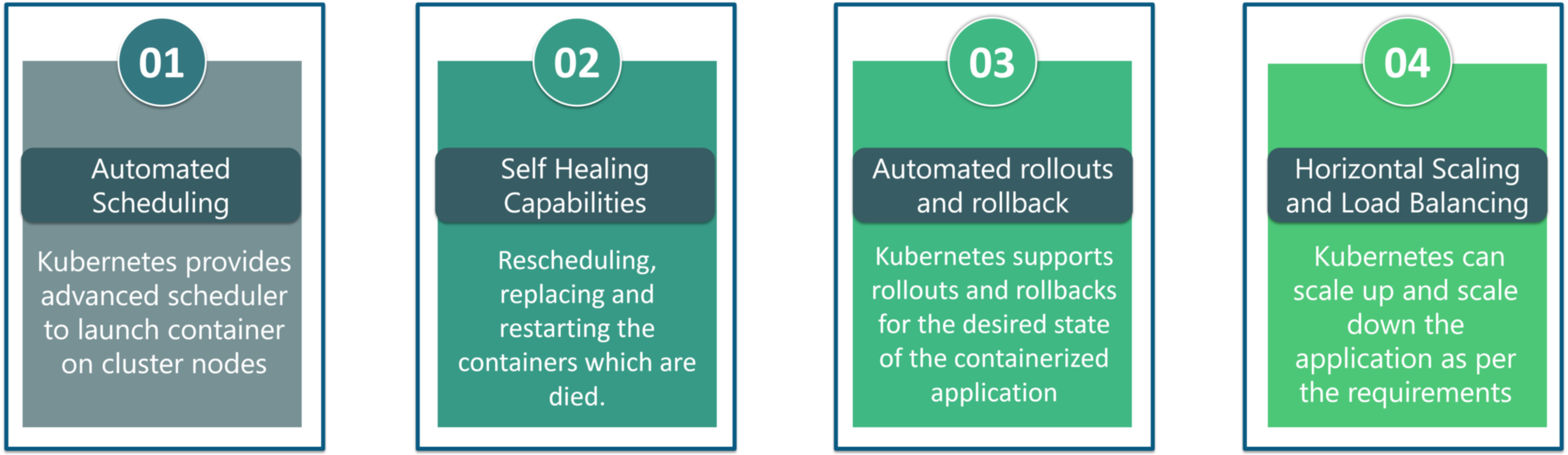
- What is Kubernetes?
- Advantages of Kubernetes
- Kubernetes Architecture
- Kubernetes Components
- Kubernetes Cluster Setup
- Kubernetes Objects
- Demo

Mithun
Technologies

What is Kubernetes?

- Kubernetes is an **orchestration engine** and **open-source platform for managing containerized applications.**
- Responsibilities include **container deployment, scaling & descaling of containers & container load balancing.**
- Actually, Kubernetes is not a replacement for Docker, But Kubernetes can be considered as a replacement for Docker Swarm, Kubernetes is significantly more complex than Swarm, and requires more work to deploy.
- Born in **Google** ,written in Go/Golang. Donated to CNCF(Cloud native computing foundation) in 2014.
- Kubernetes v1.0 was released on **July 21, 2015.**
- Current stable release v1.27

Kubernetes Features



01

Automated Scheduling

Kubernetes provides advanced scheduler to launch container on cluster nodes

02

Self Healing Capabilities

Rescheduling, replacing and restarting the containers which are died.

03

Automated rollouts and rollback

Kubernetes supports rollouts and rollbacks for the desired state of the containerized application

04

Horizontal Scaling and Load Balancing

Kubernetes can scale up and scale down the application as per the requirements

The features of Kubernetes, are as follows:

•**Automated Scheduling:** Kubernetes provides advanced scheduler to launch container on cluster nodes based on their resource requirements and other constraints, while not sacrificing availability.

•**Self Healing Capabilities:** Kubernetes allows to replaces and reschedules containers when nodes die. It also kills containers that don’t respond to user-defined health check and doesn’t advertise them to clients until they are ready to serve.

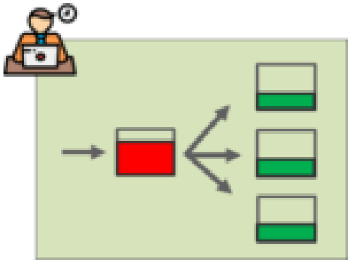
•**Automated rollouts & rollback:** Kubernetes rolls out changes to the application or its configuration while monitoring application health to ensure it doesn’t kill all your instances at the same time. If something goes wrong, with Kubernetes you can rollback the change.

•**Horizontal Scaling & Load Balancing:** Kubernetes can scale up and scale down the application as per the requirements with a simple command, using a UI, or automatically based on CPU usage.

Kubernetes Features

5. Service Discovery & Load balancing

With Kubernetes, there is no need to worry about networking and communication because Kubernetes will automatically assign IP addresses to containers and a single DNS name for a set of containers, that can load-balance traffic inside the cluster.
Containers get their own IP so you can put a set of containers behind a single DNS name for load balancing.

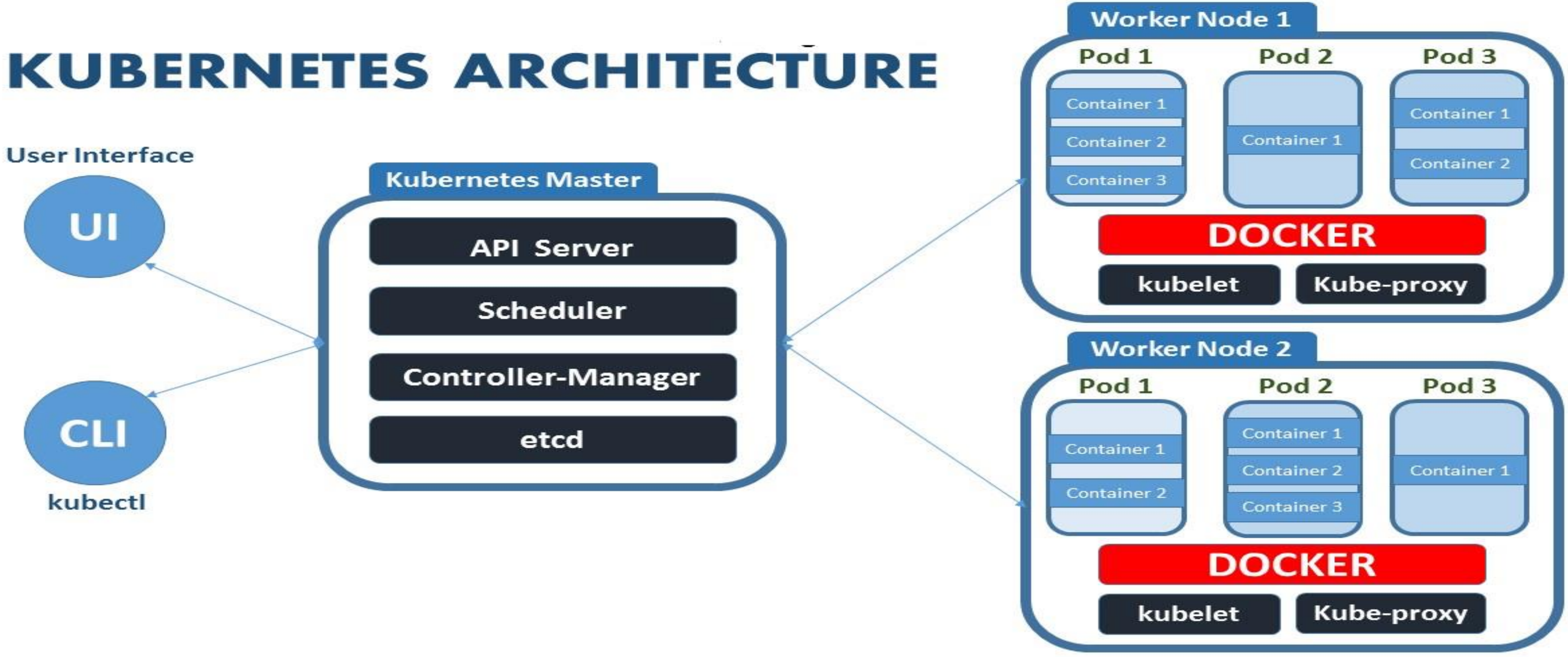


6• Storage Orchestration

With Kubernetes, you can mount the storage system of your choice. You can either opt for local storage, or choose a public cloud provider such as GCP or AWS, or perhaps use a shared network storage system such as NFS, iSCSI, etc.

Kubernetes Architecture

Kubernetes implements a cluster computing background, everything works from inside a ***Kubernetes Cluster***. This cluster is hosted by one node acting as the ‘master’ of the cluster, and other nodes as ‘nodes’ which do the actual ‘containerization‘. Below is a diagram showing the same.



Kubernetes Components

Web UI (Dashboard)

Dashboard is a web-based Kubernetes user interface. You can use Dashboard to deploy containerized applications to a Kubernetes cluster, troubleshoot your containerized application, and manage the cluster itself along with its available resources.

Kubectl

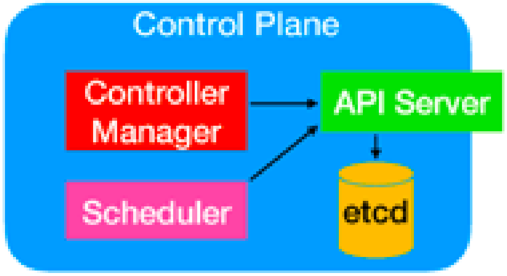
Kubectl is a command line configuration tool (CLI) for Kubernetes used to interact with master node of kubernetes. Kubectl has a config file called kubeconfig, this file has the information about server and authentication information to access the API Server.

Master Node

The master node is responsible for the management of Kubernetes cluster. It is mainly the entry point for all administrative tasks. It handles the orchestration of the worker nodes. There can be more than one master node in the cluster to check for fault tolerance.

Master Components

It has below components that take care of communication, scheduling and controllers.



API Server:

Kube API Server interacts with API, Its a frontend of the kubernetes control plane. Communication center for developers, sysadmin and other Kubernetes components

Scheduler

Scheduler watches the pods and assigns the pods to run on specific hosts.

Kubernetes Components

Controller Manager:

Controller manager runs the controllers in background which runs different tasks in Kubernetes cluster. Performs cluster-level functions (replication, keeping track of worker nodes, handling nodes failures…).

Some of the controllers are,

1. Node controller - Its responsible for noticing and responding when nodes go down.

2. Replication controllers - It maintains the number of pods. It controls how many identical copies of a pod should be running somewhere on the cluster.

3. Endpoint controllers joins services and pods together.

4. Replicaset controllers ensure number of replication of pods running at all time.

5. Deployment controller provides declarative updates for pods and replicasets.

6. Daemonsets controller ensure all nodes run a copy of specific pods.

7. Jobs controller is the supervisor process for pods carrying out batch jobs

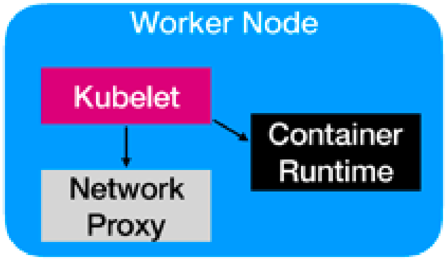
etcd

etcd is a simple distribute key value store. kubernetes uses etcd as its database to store all cluster datas. some of the data stored in etcd is job scheduling information, pods, state information and etc.

Kubernetes Components

Worker Nodes

- Worker nodes are the nodes where the application actually running in kubernetes cluster, it is also know as minion. These each worker nodes are controlled by the master node using kubelet process.
- Container Platform must be running on each worker nodes and it works together with kubelet to run the containers, This is why we use Docker engine and takes care of managing images and containers.
- We can also use other container platforms like CoreOS, Rocket.



Node Components

Kubelet

- Kubelet is the primary node agent runs on each nodes and reads the container manifests which ensures that containers are running and healthy.
- It makes sure that containers are running in a pod. The kubelet doesn’t manage containers which were not created by Kubernetes.

Kube-proxy

- kube-proxy enables the Kubernetes service abstraction by maintaining network rules on the host and performing connection forwarding.
- kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from inside or outside of your cluster.
- It helps us to have network proxy and load balancer for the services in a single worker node..
- Service is just a logical concept, the real work is being done by the “kube-proxy” pod that is running on each node.
- It redirect requests from Cluster IP(Virtual IP Address) to Pod IP.

Container Runtime

- Each node must have a container runtime, such as Docker, rkt, or another container runtime to process instructions from the master server to run containers.

Installation

Different ways to install Kubernetes

Play-with-k8s

https://labs.play-with-k8s.com.

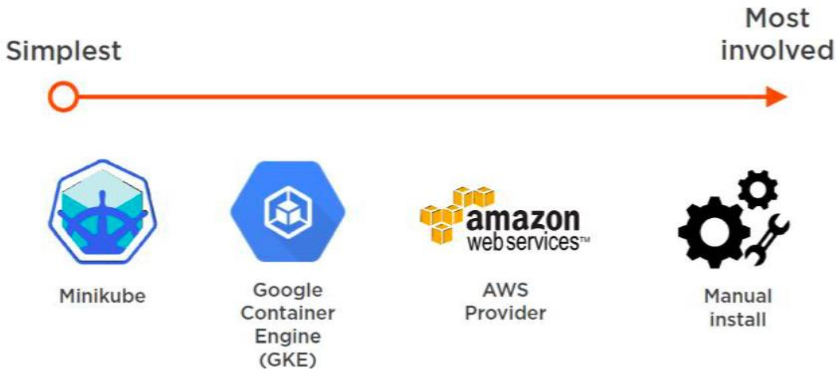
Minikube

kubeadm

Google Kubernetes Engine(GKE)

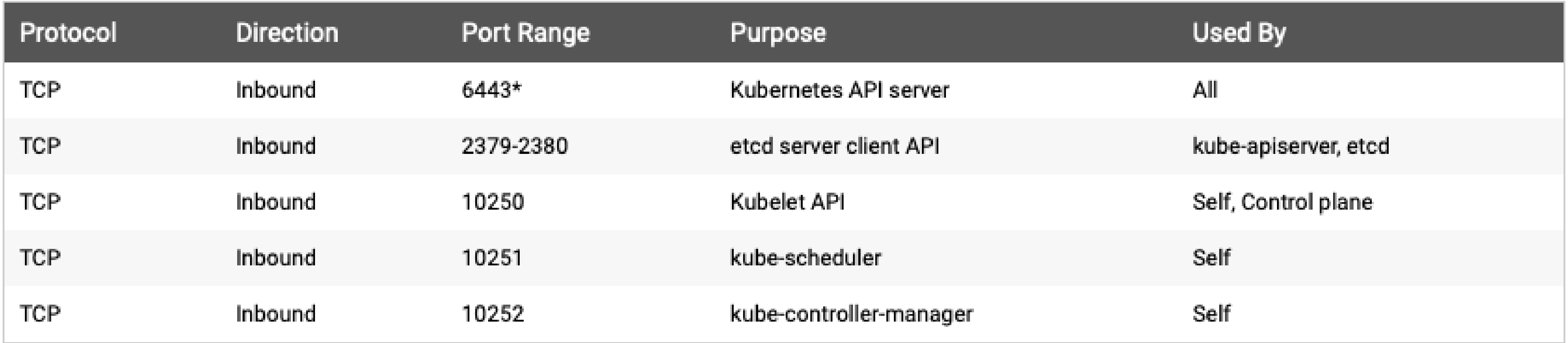
Amazon EKS

Azure Kubernetes Service (AKS)



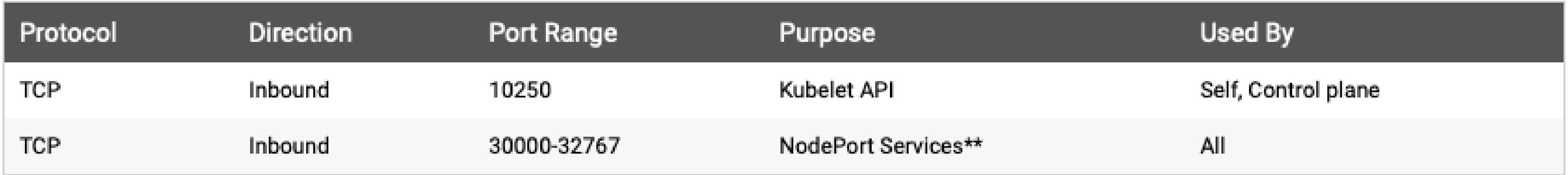
Check required ports

Control-plane node(s)



Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	6443*	Kubernetes API server	All
TCP	Inbound	2379-2380	etcd server client API	kube-apiserver, etcd
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	10251	kube-scheduler	Self
TCP	Inbound	10252	kube-controller-manager	Self

Worker node(s)



Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	30000-32767	NodePort Services**	All

Kubernetes Objects

- Kubernetes Objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster.
- A Kubernetes object is a “record of intent”–once you create the object, the Kubernetes system will constantly work to ensure that object exists.
- To work with Kubernetes objects–whether to create, modify, or delete them–you’ll need to use the Kubernetes API. When you use the kubectl command-line interface, for example, the CLI makes the necessary Kubernetes API calls for you.

The basic Kubernetes objects include:

- Pod
- Replication Controller
- ReplicaSet
- DaemonSet
- Deployment
- Service
- ClusterIP
- NodePort
- Volume
- Job

Kubernetes Objects

What is a Namespace?

You can think of a Namespace as a virtual cluster inside your Kubernetes cluster. You can have multiple namespaces inside a single Kubernetes cluster, and they are all logically isolated from each other. They can be helpful when different teams or projects share a Kubernetes cluster.

Kubernetes namespaces are a mechanism for isolating groups of resources within a single cluster. Resource names must be unique within a namespace but not across namespaces. Namespace-based scoping is only applicable to namespaced objects (such as Deployments and Services) and not to cluster-wide objects (e.g. StorageClass, Nodes, PersistentVolumes, etc).

Any number of namespaces can be created within a cluster, each logically separated from others but with the ability to communicate with each other. Namespaces cannot be nested within each other.

Default Kubernetes namespaces

Here are the four default namespaces Kubernetes creates automatically:

- **default** —a default space for objects that do not have a specified namespace.
- **kube-system** —a default space for Kubernetes system objects, such as kube-dns and kube-proxy, and add-ons providing cluster-level features.
- **kube-public** —a default space for resources available to all users without authentication.
- **kube-node-lease** —a default space for objects related to cluster scaling.

https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-organizing-with-namespaces

Kubernetes Objects

When Should You Use Multiple Kubernetes Namespaces?

In small projects or teams, where there is no need to isolate workloads or users from each other, it can be reasonable to use the default Kubernetes namespace. Consider using multiple namespaces for the following reasons:

- **Isolation**—if you have a large team, or several teams working on the same k8s cluster, you can use namespaces to create separation between projects and microservices. Activity in one namespace never affects the other namespaces.
- **Permissions**—it might be necessary to define separate permissions for different resources in your cluster. You can define separate RBAC rules for each namespace, ensuring that only authorized roles can access the resources in the namespace. This is especially important for mission critical applications, and to protect sensitive data in production deployments.
- **Resource control**—you can define resource limits at the namespace level, ensuring each namespace has access to a certain amount of CPU and memory resources. This enables separating cluster resources among several projects(teams) and ensuring each project has the resources it needs, leaving sufficient resources for other projects.
- **Performance**—Kubernetes API provides better performance when you define namespaces in the cluster. This is because the API has less items to search through when you perform specific operations.

Creating a Namespace can be done with a single command. If you wanted to create a Namespace called ‘test-ns’ you would run:

```
kubectl create namespace test-ns
```

Or you can create a YAML file and apply it just like any other Kubernetes resource.

test-ns.yaml:

```
kind: Namespace
apiVersion: v1
metadata:
  name: test-ns
  labels:
    name: test
```

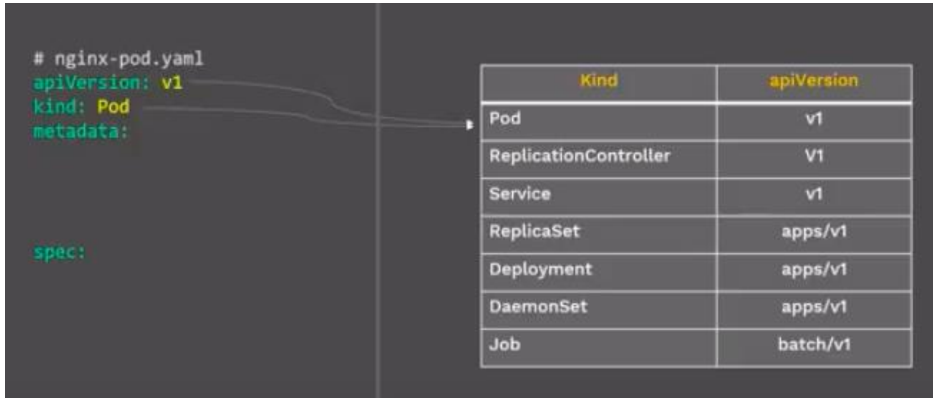
```
kubectl apply -f test-ns.yaml
```

You can see all the Namespaces with the following command

```
kubectl get namespace
```

Kubernetes Objects

POD

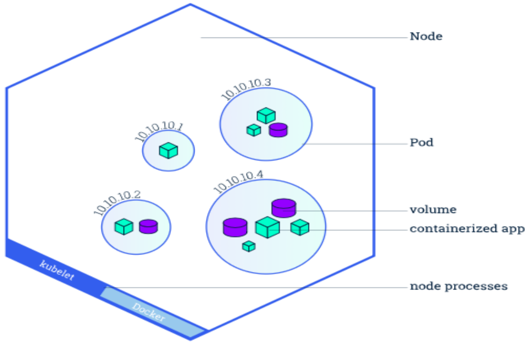


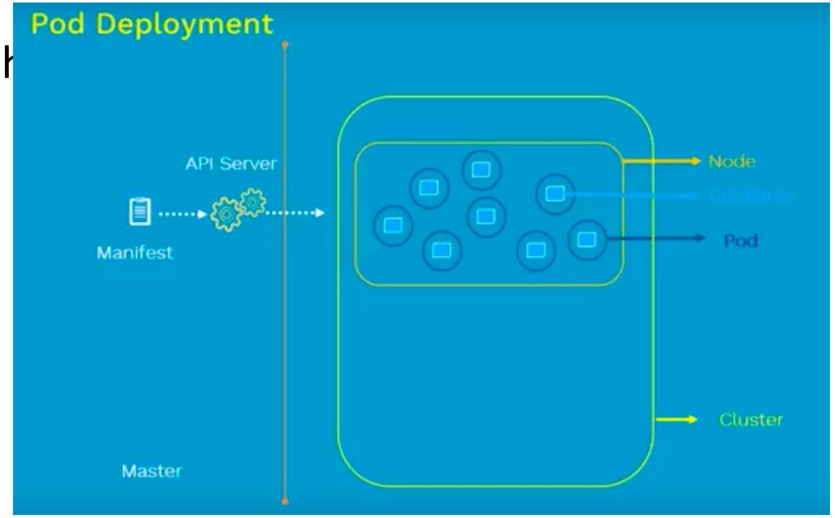
```
# nginx-pod.yaml
apiVersion: v1
kind: Pod
metadata:


spec:
```

Kind	apiVersion
Pod	v1
ReplicationController	V1
Service	v1
ReplicaSet	apps/v1
Deployment	apps/v1
DaemonSet	apps/v1
Job	batch/v1

- A Pod always runs on a **Node.**
- **A pod is the smallest building block or basic unit of scheduling in Kubernetes.**
- **In a Kubernetes cluster, a pod represents a running process.**
- **Inside a pod, you can have one or more containers. Those containers all share a unique network IP, storage, network and any other specification applied to the pod.**
- POD is a group of one or more containers which will be running on some node.
- Pods abstract network and storage away from the underlying container.
- This lets you move containers around the cluster more easily.
- Each Pod has its unique IP Address within the cluster.
- Any data saved inside the Pod will disappear with

  a pe





```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
spec:
  containers:
    - name: first-container
      image: nginx
      ports:
      - containerPort: 80
```

Command

kubectl apply –f pod.yml

Kubernetes Objects

Pod Lifecycle

- Make a Pod reuqest to API server using a local pod defination file
- The API server saves the info for the pod in ETCD
- The scheduler finds the unscheduled pod and shedules it to node.
- Kubelet running on the node, sees the pod sheduled and fires up docker.
- Docker runs the container
- The entire lifecycle state of the pod is stored in ETCD.

Pod Concepts

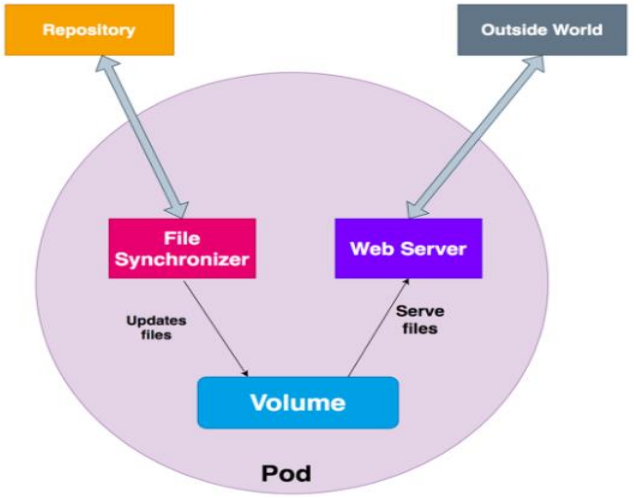
- Pod is ephemeral(lasting for a very short time) and won’t be rescheduled to a new node once it dies.
- You should not directly create/use Pod for deployment, Kubernetes have controllers like Replica Sets, Deployment, Deamon sets to keep pod alive.

Pod model types

Most often, when you deploy a pod to a Kubernetes cluster, it'll contain a single container. But there are instances when you might need to deploy a pod with multiple containers.

There are two model types of pod you can create:

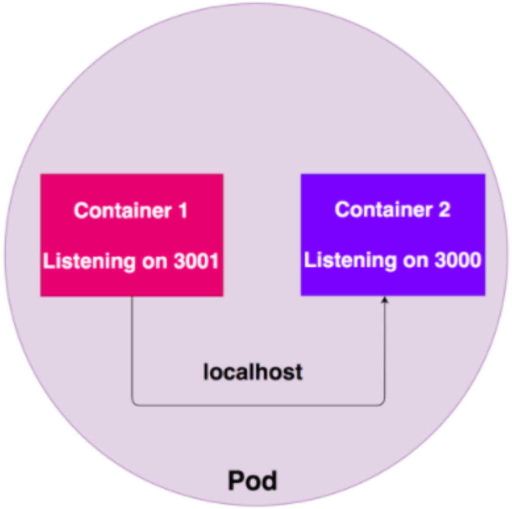
- **One-container-per-pod.** This model is the most popular. POD is the "wrapper" for a single container. Since pod is the smallest object that K8S recognizes, it manages the pods instead of directly managing the containers.
- **Multi-container-pod or Sidecar containers** In this model, a pod can hold multiple co-located containers primary container **And** utility container that helps or enhances how an application functions (examples of sidecar containers are log shippers/watchers and monitoring agents).



Multi Container Pod

Shared Network Namespace

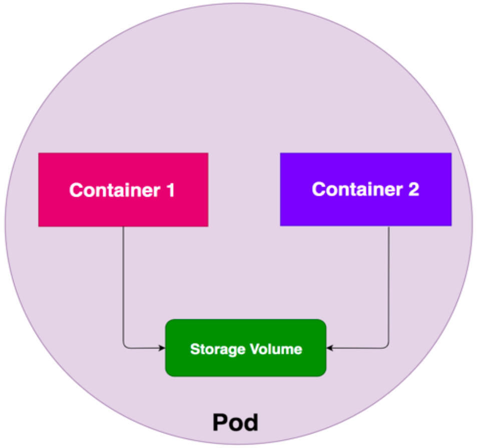
All the containers in the pod share the same network namespace which means all containers can communicate with each other on the **localhost**.



Shared Network Space

Shared Storage Volumes

All the containers in a POD can have the same volume mounted so that they can communicate with each other by reading and modifying files in the storage volume.



Shared Storage Volumes

Static Pods

- Static Pods are managed directly by the kubelet and the API server does not have any control over these pods. The kubelet is responsible to watch each static Pod and restart it if it crashes. The static Pods running on a node are visible on the API server but cannot be controlled by the API Server. Static Pod does not have any associated replication controller, kubelet service itself watches it and restarts it when it crashes. There is no health check for static pods.

- The main use for static Pods is to run a self-hosted control plane: in other words, using the kubelet to supervise the individual control plane components.
- Static Pods are always bound to one Kubelet on a specific node.

Kubernetes Labels and Selectors

Labels

When One thing in k8s needs to find another things in k8s, it uses labels.

Labels are key/value pairs attached to Object

You can make your own and apply it.

it’s like tag things in kubernetes

For e.g.

```
 labels:
   app: nginx
   role: web
   env: dev
```

Selectors

Selectors use the label key to find a collection of objects matched with same value

It’s like Filter, Conditions and query to your labels

For e.g.

```
selectors:
  env = dev
  app != db
  release in (1.3,1.4)
```

Labels and Selectors are used in many places like Services, Deployment and we will see now in Replicasets.

Kubernetes Objects

Service

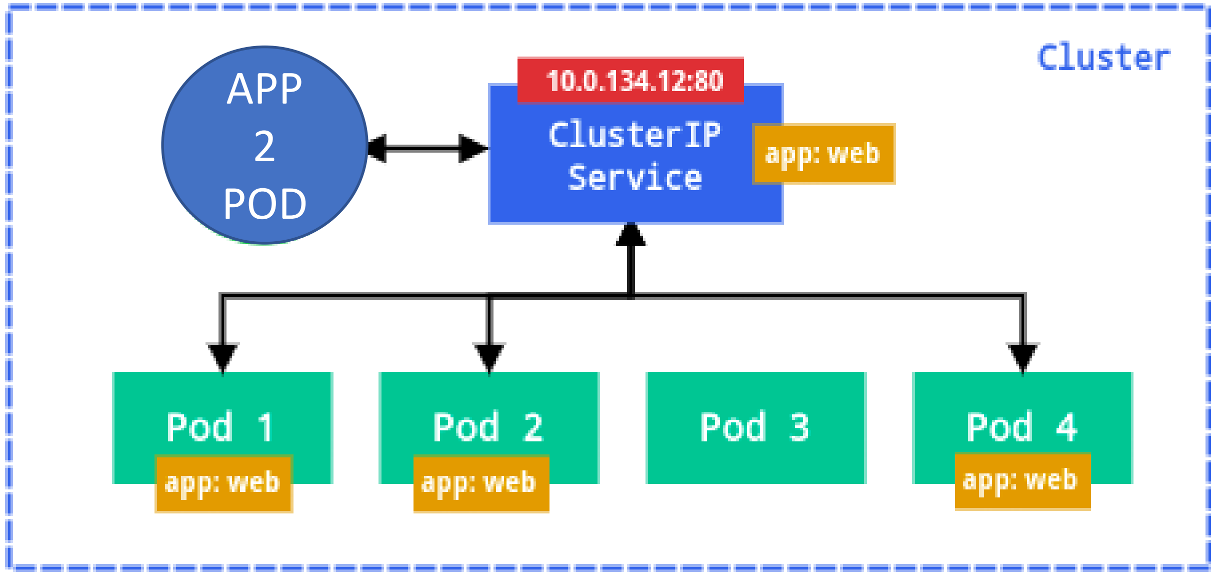
A service is responsible for making our Pods discoverable inside the network or exposing them to the internet. A Service identifies Pods by its LabelSelector.

Types of services available:

ClusterIP – Exposes the service on a cluster-internal IP. Service is only reachable from within the cluster. This is the default Type.

- When we create a service we will get one Virtual IP (Cluster IP) it will get registered to the DNS(kube-dns). Using this Other PODS can find and talk the pods of this service using service name.
- Service is just a logical concept, the real work is being done by the “kube-proxy” pod that is running on each node.
- It redirect requests from Cluster IP(Virtual IP Address) to Pod IP.

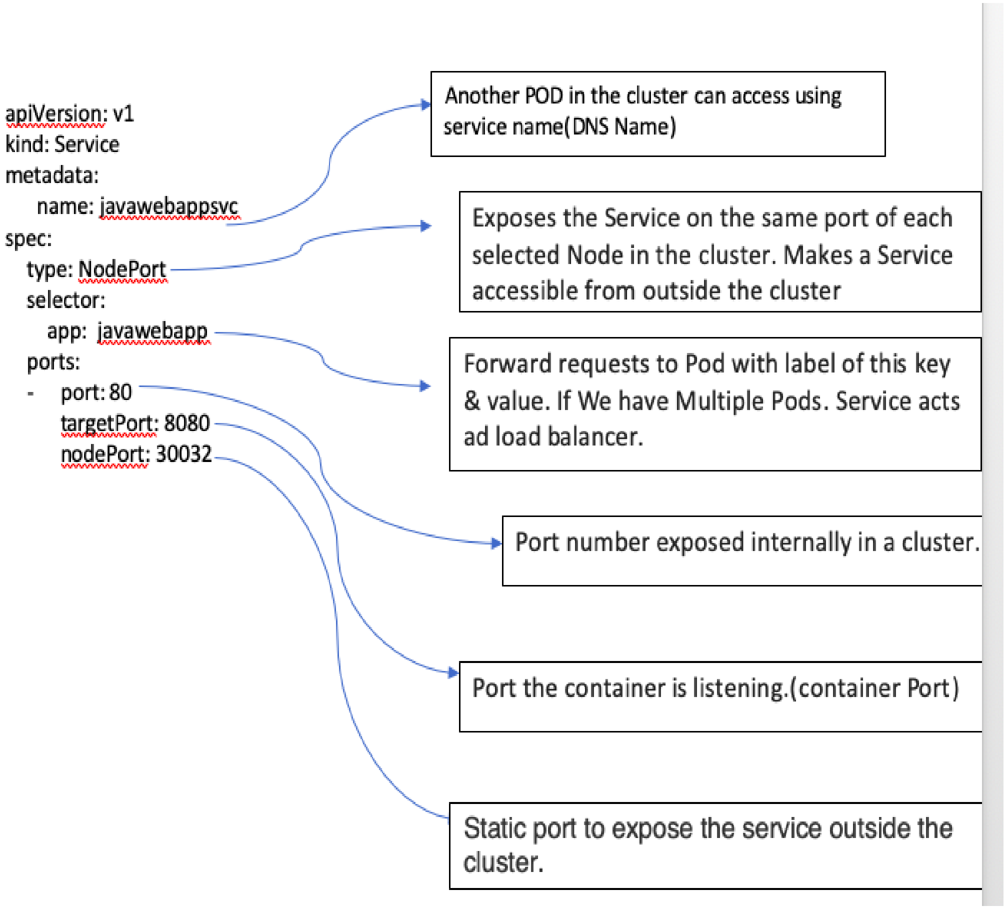
```
# Service Cluster IP
apiVersion: v1
kind: Service
metadata:
  name: nginxsvc
spec:
  selector:
    app: nginx
  type: ClusterIP
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
```



Kubernetes Objects

NodePort – Exposes the service on each Node’s IP at a static port. A ClusterIP service, to which the NodePort service will route, is automatically created. You’ll be able to contact the NodePort service, from outside the cluster, by using “*<NodeIP>:<NodePort>*”.

```
apiVersion: v1
kind: Service
metadata:
  name: javawebappsvc
spec:
  selector:
    app: javawebapp
  type: NodePort
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30032
```



```
apiVersion: v1
kind: Service
metadata:
   name: javawebappsvc
spec:
  type: NodePort
  selector:
    app:  javawebapp
  ports:
  -    port: 80
       targetPort: 8080
       nodePort: 30032
```

Another POD in the cluster can access using service name(DNS Name)

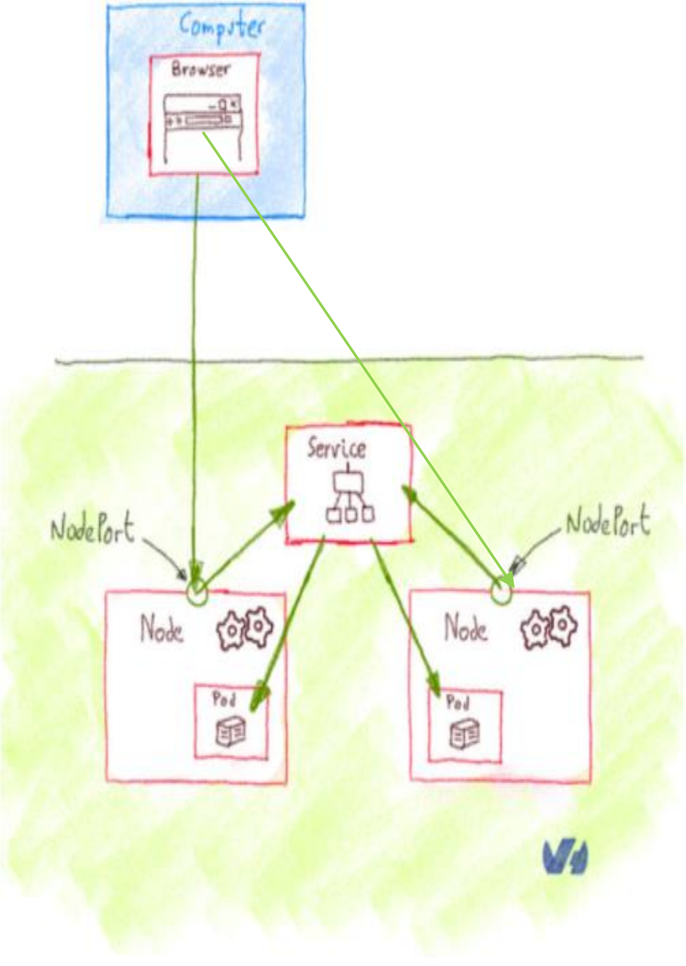
Exposes the Service on the same port of each selected Node in the cluster. Makes a Service accessible from outside the cluster

Forward requests to Pod with label of this key & value. If We have Multiple Pods. Service acts ad load balancer.

Port number exposed internally in a cluster.

Port the container is listening.(container Port)

Static port to expose the service outside the cluster.



NodePort

Note: If we don’t define nodePort value for NodePort Service. K8’s will randomly Allocate a nodePort with in 30000—32767.

Kubernetes Objects

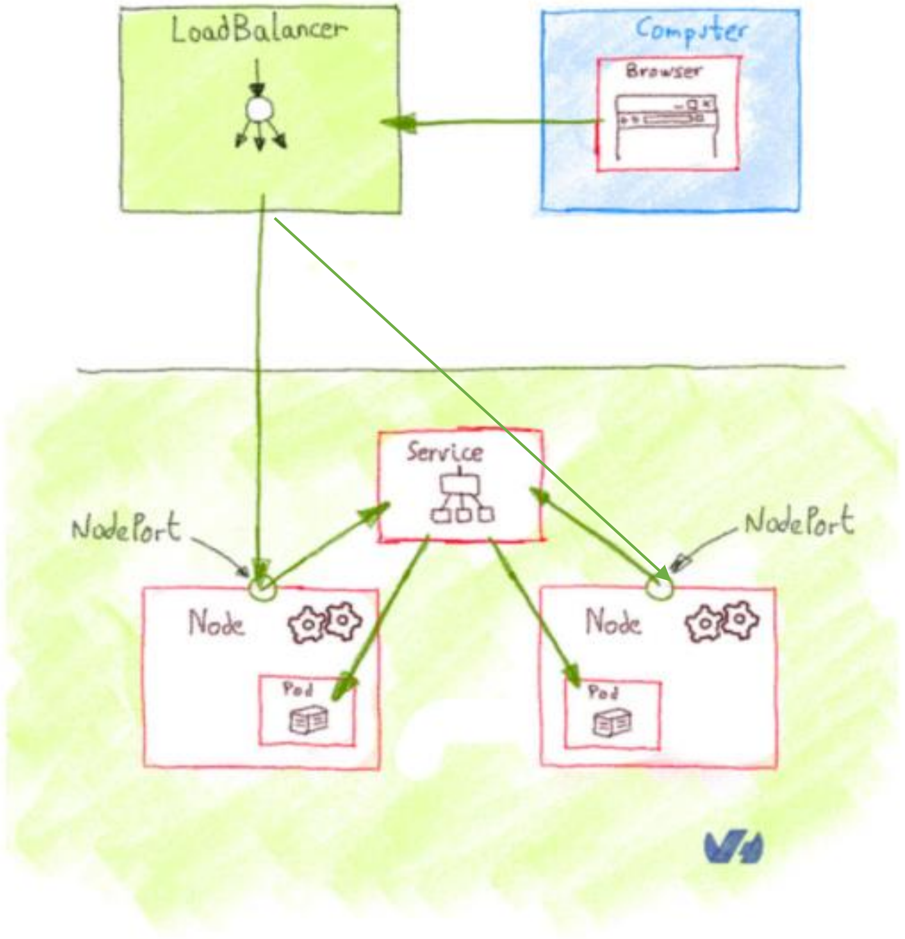
LoadBalancer – Exposes the service externally using a cloud provider's load balancer. NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.

Mithun Technologies

If you are using a custom Kubernetes Cluster (using minikube, kubeadm or the like). In this case, there is no LoadBalancer integrated (unlike AWS EKS or Google Cloud,KOPS,AKS). With this default setup, you can only use NodePort.

Configure Load Balancer Externally.

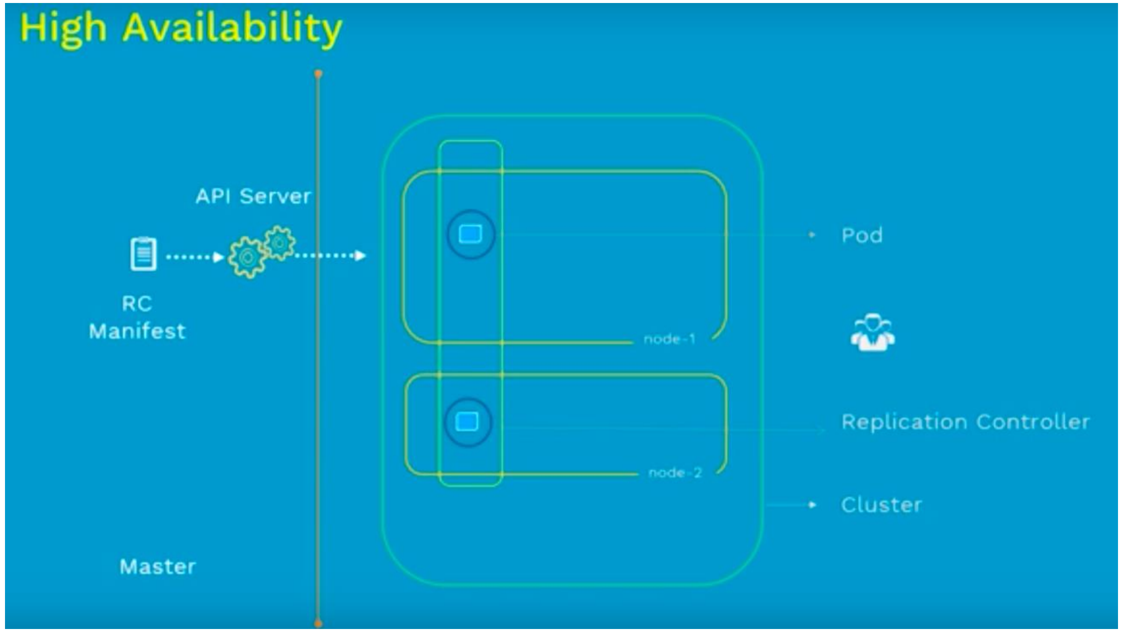
```
apiVersion: v1
kind: Service
metadata:
  name: javawebappsvc
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: javawebapp
  type: LoadBalancer
```



LoadBalancer

Replication Controller

- Replication Controller is one of the key features of Kubernetes, which is responsible for managing the pod lifecycle. It is responsible for making sure that the specified number of pod replicas are running at any point of time.
- A Replication Controller is a structure that enables you to easily create multiple pods, then make sure that that number of pods always exists. If a pod does crash, the Replication Controller replaces it.
- Replication Controllers and PODS are associated with labels.
- Creating “RC” with count of 1 ensure that a POD is always available.



Command

```
kubectl apply –f rc.yml
```

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx-rc
spec:
  replicas: 2
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

Mithun Technologies

ReplicaSet

- ReplicaSet is the next-generation Replication Controller.
- The only difference between a *ReplicaSet* and a *Replication Controller* right now is the selector support.
- ReplicaSet supports the new set-based selector requirements as described in the labels user guide whereas a Replication Controller only supports equality-based selector requirements.

```
kubectl apply –f rs.yml
kubectl scale rs nginx-replicaset --replicas 3
```

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx-rs-pod
    matchExpressions:
    - key: env
      operator: In
      values:
      - dev
  template:
    metadata:
      labels:
        app: nginx-rs-pod
        env: dev
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

DaemonSet

A *DaemonSet* make sure that all or some kubernetes Nodes run a copy of a Pod.

When a new node is added to the cluster, a Pod is added to it to match the rest of the nodes and when a node is removed from the cluster, the Pod is garbage collected.

Deleting a DaemonSet will clean up the Pods it created.

```
# Daemon Set
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-ds
spec:
  selector:
    matchLabels:
      app: nginx-pod
  template:
    metadata:
      name: nginx-pod
      labels:
        app: nginx-pod
    spec:
      nodeSelector:
        name: node1
      containers:
        - name: webserver
          image: nginx
          ports:
          - containerPort: 80
```

Command

```
kubectl apply –f daemonset.yml
```

Kubernetes Objects

Deployment

In Kubernetes, Deployment is the recommended way to deploy Pod or RS, simply because of the advance features it comes with.

Below are some of the key benefits.

- Deploy a RS.
- Updates pods (PodTemplateSpec).
- Rollback to older Deployment versions.
- Scale Deployment up or down.
- Pause and resume the Deployment.
- Use the status of the Deployment to determine state of replicas.
- Clean up older RS that you don’t need anymore.

```
kubectl scale deployment [DEPLOYMENT_NAME] --replicas [NUMBER_OF_REPLICAS]
kubectl rollout status deployment nginx-deployment
kubectl get deployment
kubectl set image deployment nginx-deployment nginx-container=nginx:latest --record
kubectl get replicaset
kubectl rollout history deployment nginx-deployment
kubectl rollout history deployment nginx-deployment --revision=1
kubectl rollout undo deployment nginx-deployment --to-revision=1
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  strategy:
    type: ReCreate
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Kubernetes Objects

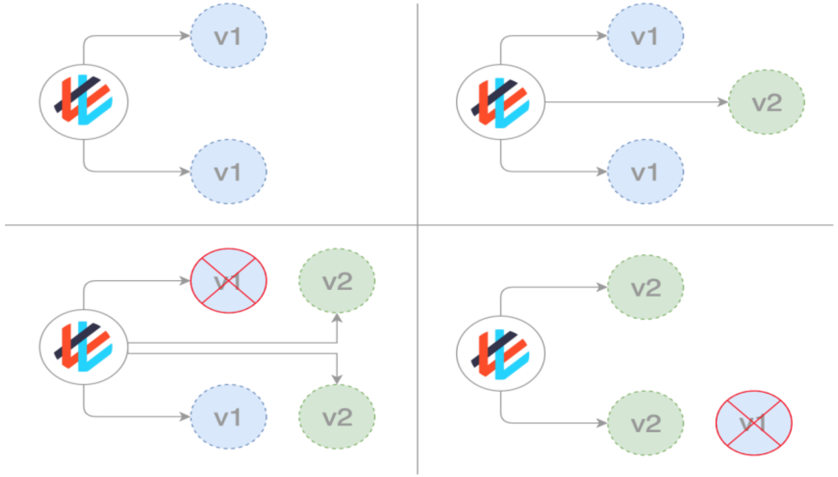
Deployment Strategies

There are several different types of deployment strategies you can take advantage of depending on your goal.

Rolling Deployment

The rolling deployment is the standard default deployment to Kubernetes. It works by slowly, one by one, replacing pods of the previous version of your application with pods of the new version without any cluster downtime.

A rolling update waits for new pods to become ready before it starts scaling down the old ones. If there is a problem, the rolling update or deployment can be aborted without bringing the whole cluster down. In the YAML definition file for this type of deployment, a new image replaces the old image.



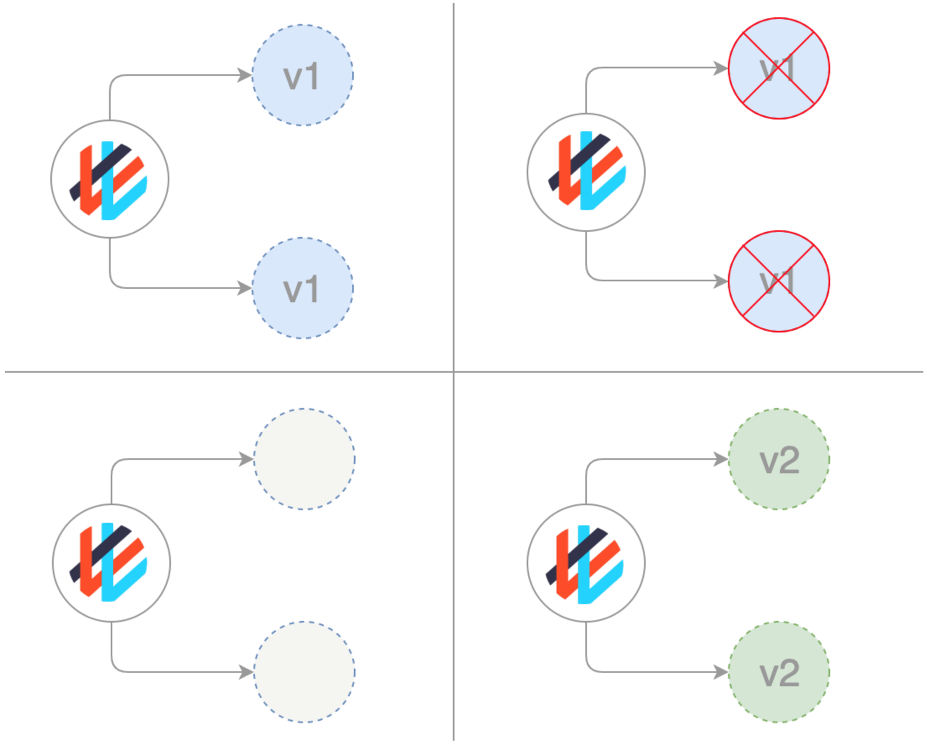
```
# Deployments  Rolling Update
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  strategy:
    type: RollingUpdate
    rollingUpdate:
       maxSurge: 1
       maxUnavailable: 1
  minReadySeconds: 30
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      name: nginx-pod
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Kubernetes Objects

Deployment Strategies

Recreate

- In this type of very simple deployment, all of the old pods are killed all at once and get replaced all at once with the new ones.



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  strategy:
  type: Recreate
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Mithun Technologies

Kubernetes Objects

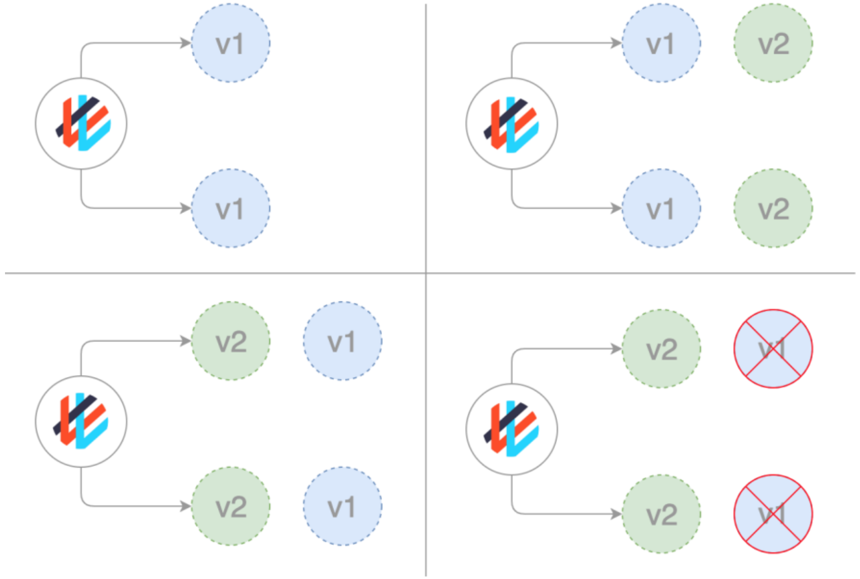
Deployment Strategies

Blue/ Green (or Red / Black) deployments

In a blue/green deployment strategy (sometimes referred to as red/black) the old version of the application (green) and the new version (blue) get deployed at the same time. When both of these are deployed, users only have access to the green; whereas, the blue is available to your QA team for test automation on a separate service or via direct port-forwarding.

After the new version has been tested and is signed off for release, the service is switched to the blue version with the old green version being scaled down:

```
apiVersion: v1
kind: Service
metadata:
  name: awesomeapp
spec:
  selector:
    app: awesomeapp
    version: "02"
```



Mithun Technologies

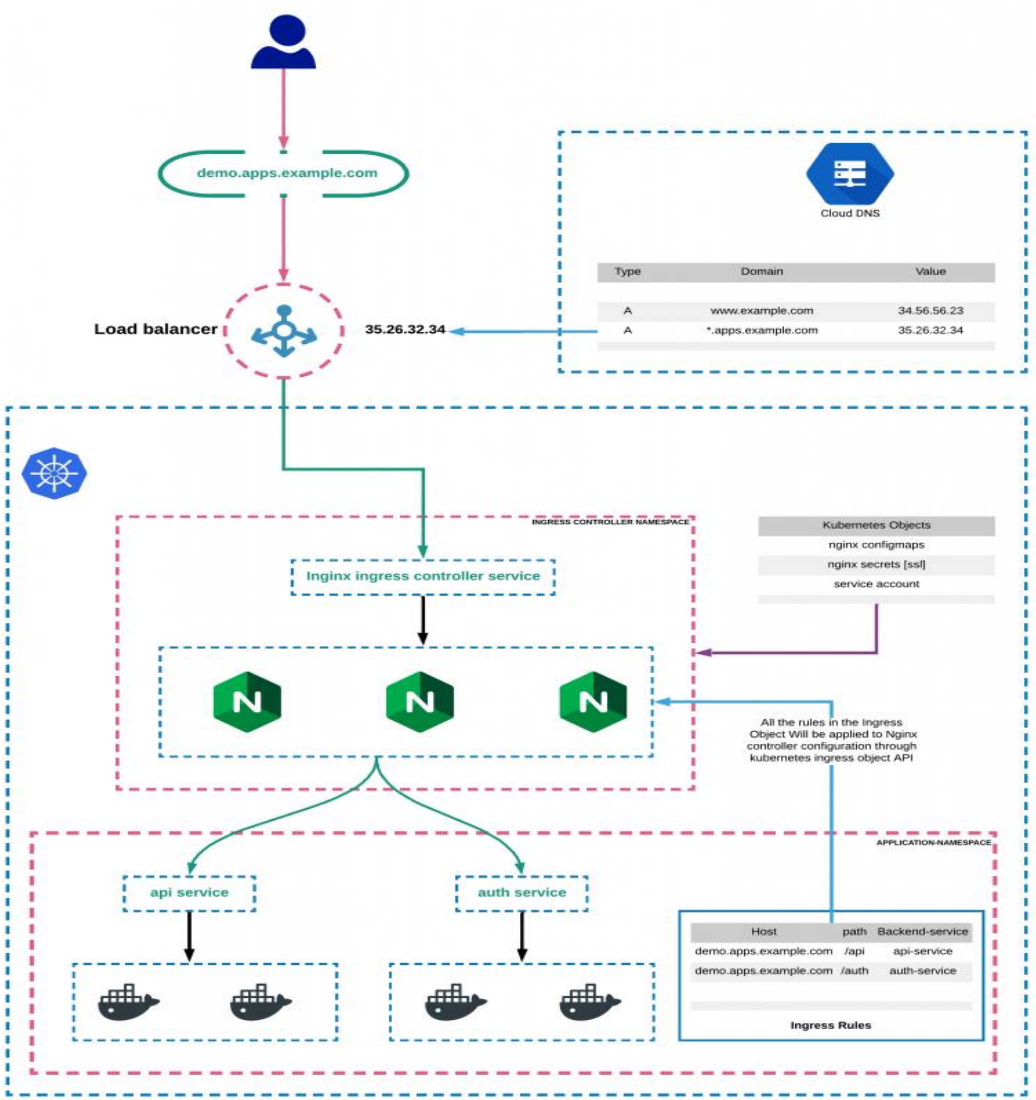
Kubernetes Objects

Ingress *– Kubernetes Ingress is a resource to add rules for routing traffic from external sources to the services in the kubernetes cluster.*

Kubernetes Ingress:
Kubernetes Ingress is a native kubernetes resource where you can have rules to route traffic from an external source to service endpoints residing inside the cluster. It requires an ingress controller for routing the rules specified in the ingress object.

Kubernetes Ingress Controller
Ingress controller is typically a proxy service deployed in the cluster. It is nothing but a kubernetes deployment exposed to a service. Following are the ingress controllers available for kubernetes.

Following are the ingress controllers available for kubernetes.
Nginx Ingress Controller (Community & From Nginx Inc)
Traefik
HAproxy
Contour
GKE Ingress Controller


demo.apps.example.com
Load balancer
35.26.32.34
Cloud DNS
INGRESS CONTROLLER NAMESPACE
Inginx ingress controller service
All the rules in the Ingress Object Will be applied to Nginx controller configuration through kubernetes ingress object API
APPLICATION-NAMESPACE
api service
auth service

Type	Domain	Value
A	www.example.com	34.56.56.23
A	*.apps.example.com	35.26.32.34

Kubernetes Objects
nginx configmaps
nginx secrets [ssl]
service account

Host	path	Backend-service
demo.apps.example.com	/api	api-service
demo.apps.example.com	/auth	auth-service

Ingress Rules

Kubernetes Volumes

A Container's file system lives only as long as the Container does. So when a Container terminates and restarts, filesystem changes are lost. Volumes in Kubernetes are very easy to manage. It is basically a directory that gets mounted to a pod. After telling the container to use Volumes for storing evergreen information, you can safely modify the pods without ever losing your data.

Kubernetes supports many types of volumes, and a Pod can use any number of them simultaneously. For more consistent storage that is independent of the Container, we can use a Volume. This is especially important for stateful applications, such as databases.

To use a volume, a Pod specifies what volumes to provide for the Pod
(the .spec.volumes field) and where to mount those into Containers
(the .spec.containers.volumeMounts field) as shown.

Kubernetes supports several types of Volumes:

awsElasticBlockStore
azureDisk
azureFile
configMap
emptyDir
gcePersistentDisk
gitRepo (deprecated)
hostPath
nfs
persistentVolumeClaim
secret

```
# Mongo host path rc
apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: mongo
  name: mongo-controller
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: mongo
    spec:
      containers:
      - image: mongo
        name: mongo
        ports:
        - name: mongo
          containerPort: 27017
          hostPort: 27017
        volumeMounts:
        - name: mongo-persistent-storage
          mountPath: /data/db
      volumes:
      - name: mongo-persistent-storage
        hostPath:
         path: /tmp/dbbackup
```

Kubernetes Volumes

Persistent Volumes

Persistent Volumes are simply a piece of storage in your cluster. Similar to how you have a disk resource in a server, a persistent volume provides storage resources for objects in the cluster. At the most simple terms you can think of a PV as a disk drive. It should be noted that this storage resource exists independently from any pods that may consume it. Meaning, that if the pod dies, the storage should remain intact assuming the claim policies are correct. Persistent Volumes are provisioned in two ways, Statically or Dynamically.

Static Volumes – A static PV simply means that some k8s administrator provisioned a persistent volume in the cluster and it's ready to be consumed by other resources.

Dynamic Volumes – In some circumstances a pod could require a persistent volume that doesn't exist. In those cases it is possible to have k8s provision the volume as needed if storage classes were configured to demonstrate where the dynamic PVs should be built. This post will focus on static volumes for now.

Persistent Volume Claims

Pods that need access to persistent storage, obtain that access through the use of a Persistent Volume Claim. A PVC, binds a persistent volume to a pod that requested it.

When a pod wants access to a persistent disk, it will request access to the claim which will specify the size , access mode and/or storage classes that it will need from a Persistent Volume. Indirectly the pods get access to the PV, but only through the use of a PVC.

Claim Policies

We also reference claim policies earlier. A Persistent Volume can have several different claim policies associated with it including:

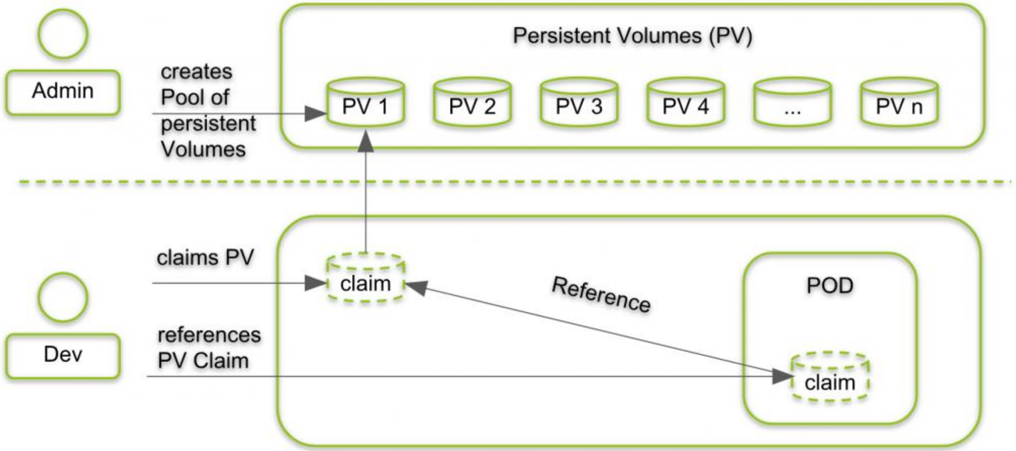
Retain – When the PVC is deleted, the PV still exists and the volume is considered "released". But it is not yet available for another claim because the previous claimant's data remains on the volume. An administrator we can manually reclaim the volume.

Recycle – When the claim is deleted the volume remains but performs a basic scrub(delete data from storage) (rm -rf /volume/*) on the volume and makes it available again for a new claim.

Delete – When the claim is deleted, it removes both the Persistent Volume object from Kubernetes, as well as the associated storage.

The claim policy (associated at the PV and not the PVC) is responsible for what happens to the data on when the claim has been deleted.

Kubernetes Volumes



ConfigMaps and Secrets
Application’s configuration can change between environments (e.g. development, staging, production, etc.) and that you want your applications to be portable. It is a good practice to make container images as reusable as possible. The same image should be able to be used for development, staging, and production.

Thus, you should store the config(like db connection details, passwords, API Keys ..etc) outside of the application itself.

We externalize things that can change and this in turn helps keep our applications portable. This is critical in the Kubernetes world where our applications are packaged as Docker images.

With Docker and containers this means we should try to keep configuration out of the container image. Developer should not hard code configuration details instead they can refer/use **Environment Variables.**

Docker makes it easy to build containers with environment variables baked in. In your Dockerfile, you can specify them with the ENV directive.You can also override the environment variables when running on the command line.

ConfigMaps and Secrets

An app’s *config* is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes database connections details and credentials or cloud credentials ..etc.

Kubernetes allows you to provide configuration information to our applications through ConfigMaps or secret resources. The main differentiator between the two is the way a pod stores the receiving information and how the data is stored in the etcd data store.

Both, ConfigMaps and Secrets store data as a key value pair. The major difference is, **Secrets store data in base64** format meanwhile **ConfigMaps store data in a plain text**.

If you have some critical data like, keys, passwords, service accounts credentials, db connection string, etc then you should always go for **Secrets** rather than Configs.

And if you want to do some application configuration using environment variables which you don't want to keep secret/hidden like, app base platform url, etc then you can go for **ConfigMaps.**

Sample Manifest Links: https://github.com/MithunTechnologiesDevOps/Kubernates-Manifests/blob/master/mysql-deployment-configmap.yml
https://github.com/MithunTechnologiesDevOps/Kubernates-Manifests/blob/master/mysql-deployment-configSecret.yml

Secret:

docker-registry

This is used by the kubelet when passed in a pod template if there is an **imagePullsecret** to provide the credentials needed to authenticate to a private Docker registry:

```
kubectl create secret docker-registry registryKey --docker-server <PrivateRepoURL> --docker-username <userName> --docker-password <password> --docker-email <email>
```

https://opensource.com/article/19/6/introduction-kubernetes-secrets-and-configmaps
https://medium.com/google-cloud/kubernetes-configmaps-and-secrets-68d061f7ab5b
https://www.oreilly.com/library/view/kubernetes-best-practices/9781492056461/ch04.html

Liveness And Readiness Probes

Liveness and Readiness probes are used to control the health of an application running inside a Pod’s container.

Liveness Probe

- Suppose that a Pod is running our application inside a container, but due to some reason let’s say memory leak, cpu usage, application deadlock etc the application is not responding to our requests, and stuck in error state.
- Liveness probe checks the container health as we tell it do, and if for some reason the liveness probe fails, it restarts the container.

Readiness Probe

- This type of probe is used to detect if a container is ready to accept traffic. You can use this probe to manage which pods are used as backends for load balancing services. If a pod is not ready, it can then be removed from the list of load balancers.

Manifest File Link:

https://github.com/MithunTechnologiesDevOps/Kubernates-Manifests/blob/master/liveness_readiness_probes_example.yml

https://docs.okd.io/latest/dev_guide/application_health.html

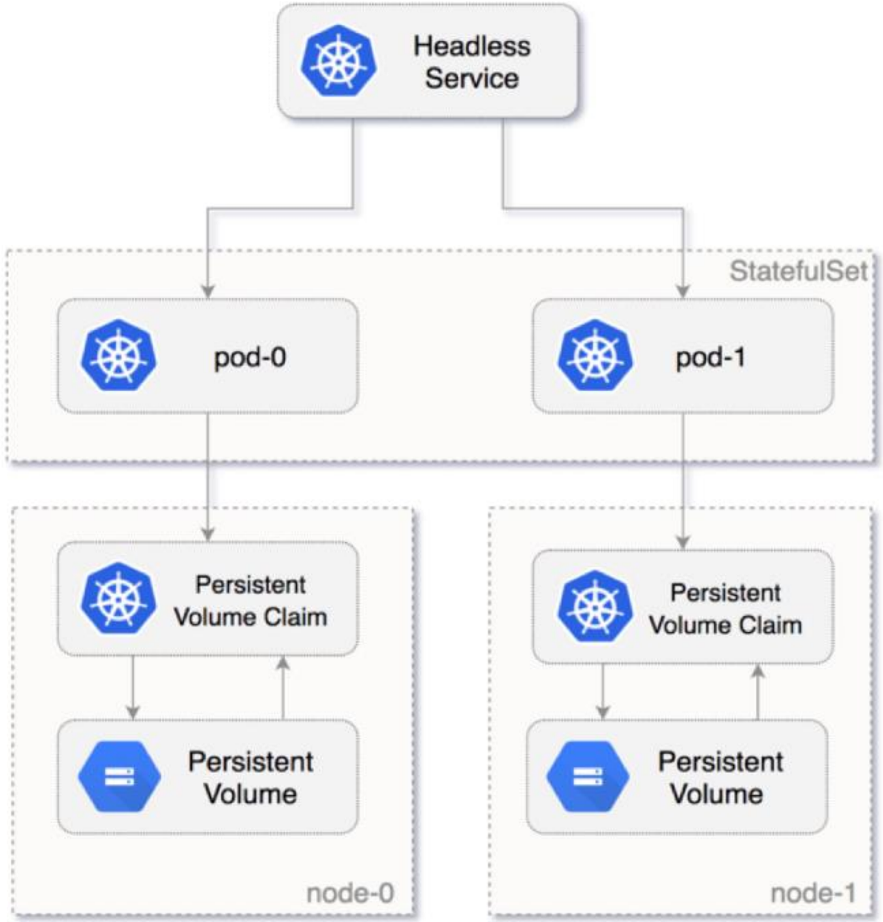
https://medium.com/spire-labs/utilizing-kubernetes-liveness-and-readiness-probes-to-automatically-recover-from-failure-2fe0314f2b2e

https://www.weave.works/blog/resilient-apps-with-liveness-and-readiness-probes-in-kubernetes

Stateful Sets

Deployments are usually used for stateless applications. However, you can save the state of deployment by attaching a Persistent Volume to it and make it stateful, but all the pods of a deployment will be sharing the same Volume and data across all of them will be same.

- SatefulSet is a Kubernetes resource used to manage stateful applications. It manages the deployment and scaling of a set of Pods, and provides guarantee about the ordering and uniqueness of these Pods.
- StatefulSet is also a Controller but unlike Deployments, it doesn’t create ReplicaSet rather itself creates the Pod with a unique naming convention. e.g. If you create a StatefulSet with name **mongo,** it will create a pod with name **mongo-0,** and for multiple replicas of a statefulset, their names will increment like **mongo-0**, **mongo-1**, **mongo-2, etc**
- Every replica of a stateful set will have its own state, and each of the pods will be creating its own PVC(Persistent Volume Claim). So a statefulset with 3 replicas will create 3 pods, each having its own Volume, so total 3 PVCs.
- By far the most common way to run a database, StatefulSets is a feature fully supported as of the Kubernetes 1.9 release. Using it, each of your pods is guaranteed the same network identity and disk across restarts, even if it's rescheduled to a different physical machine.



Stateful Sets

- **StatefulSet Deployments provide:**
- **Stable, unique network identifiers:** Each pod in a StatefulSet is given a hostname that is based on the application name and increment. For example, mongo-1, mongo-2 and mongo-3 for a StatefulSet named “mongo” that has 3 instances running.
- **Stable, persistent storage:** Each and every pod in the cluster is given its own persistent volume based on the storage class defined, or the default, if none are defined. Deleting or scaling down pods will not automatically delete the volumes associated with them- so that the data persists. you could scale the StatefulSet down to 0 first, prior to deletion of the unused pods.
- **Ordered, graceful deployment and scaling:** Pods for the StatefulSet are created and brought online in order, from 1 to n, and they are shut down in reverse order to ensure a reliable and repeatable deployment and runtime. The StatefulSet will not even scale until all the required pods are running, so if one dies, it recreates the pod before attempting to add additional instances to meet the scaling criteria.
- **Ordered, automated rolling updates:** StatefulSets have the ability to handle upgrades in a rolling manner where it shuts down and rebuilds each node in the order it was created originally, continuing this until all the old versions have been shut down and cleaned up. Persistent volumes are reused, and data is automatically migrated to the upgraded version.

Mithun Technologies

Questions ?